

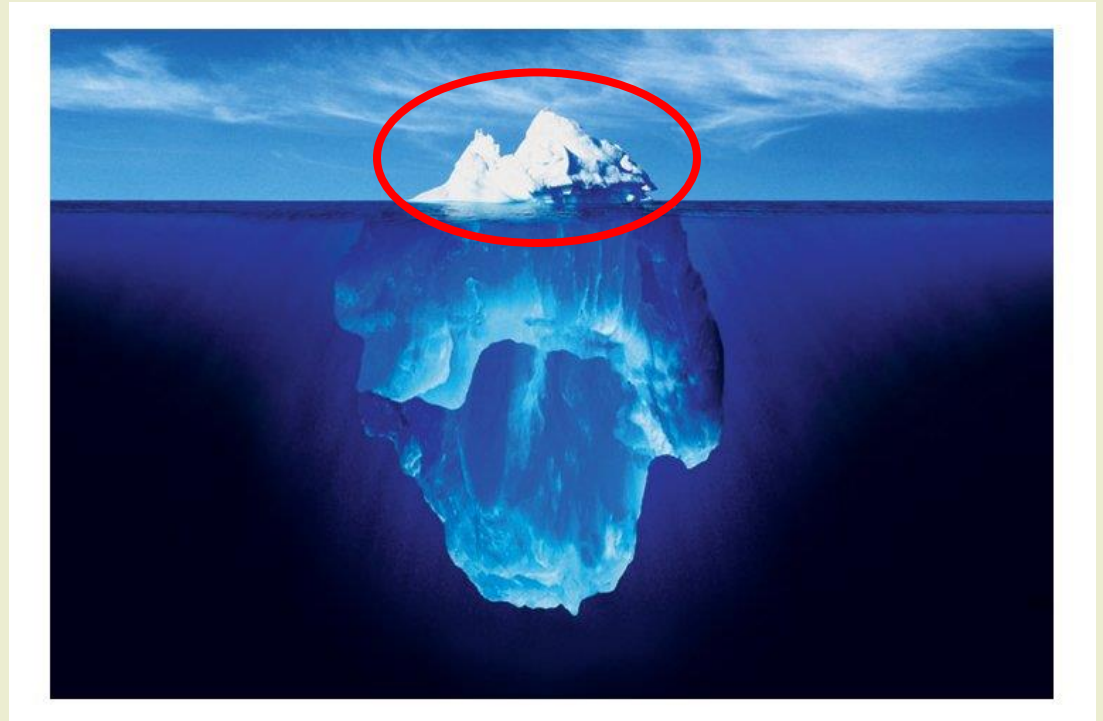
SCRIPTING AND TEXT MANIPULATION

DANIEL JUMPER & MIKE BEAUMIER

Part 1 of 2:
Text
manipulation
with reg-ex,
sed and awk

SCOPE OF TALK

- Broad and shallow



OVERVIEW

- Text Manipulation
 - Regular Expressions
 - sed
 - awk

- Scripts
 - Perl – Higher level; more structured programming
 - Bash – More simple, easy to use command line tools

Part 2

REGULAR EXPRESSIONS



THE REGULAR EXPRESSION

- What is a Regular Expression?
 - Basic definition: A set of characters that specify a pattern
 - In practice: Powerful tool for searching/matching within strings
- What is it good for?
 - Useful in context of other tools
 - Code, scripts, commands, text editing, etc...
 - Sophisticated find (and replace)
 - Extract part of a string
 - Work with strings with varying/unknown format or content
- Practice/References here:
 - <http://regexone.com/> - interactive regex tutor
 - <http://regexpal.com/> - test your regex against a sample string
 - <https://www.cs.tut.fi/~jkorpela/perl/regexp.html> - quick reference
 - <http://www.grymoire.com/Unix/Regular.html> - detailed guide

REGEX BASICS

- There are two categories of special characters used to build regular expressions, in conjunction with literal-match characters
 - Meta Characters
 - These characters define relationships between other characters in a regular expression, as well as define how many times a character or expression should be matched before flagging a ‘successful regex match’
 - Special Characters
 - These characters represent single characters of a certain character class – such as white-space (tab, space or newline), numeric characters, alphabetical characters, uppercase characters and lowercase characters.

REGEX – SPECIAL CHARACTERS

Regex	What It Matches
<code>\t</code>	tab
<code>\n</code>	newline
<code>\r</code>	return (CR)
<code>\xhh</code>	character with hex. code hh
<code>\b</code>	“word” boundary
<code>\B</code>	not a “word” boundary
<code>\w</code>	matches any single character classified as a “word” character (alphanumeric or “_”)
<code>\W</code>	matches any non-“word” character
<code>\s</code>	matches any whitespace character (space, tab, newline)
<code>\S</code>	matches any non-whitespace character
<code>\d</code>	matches any digit character, equiv. to [0-9]
<code>\D</code>	matches any non-digit character

REGEX – META CHARACTERS

Regex	What It Means
^	beginning of string, or ‘not’ inside []
\$	end of string
.	any character except newline
*	match 0 or more times
+	match 1 or more times
?	match 0 or 1 times; or: shortest match
 	alternative
()	grouping; “storing”
[]	set of characters
{ }	repetition modifier
\	quote or special

REGEX – REPEATED CHARACTERS

a*	zero or more a's
a+	one or more a's
a?	zero or one a's (i.e., optional a)
a{m}	exactly m a's
a{m,}	at least m a's
a{m,n}	at least m but at most n a's

PUTTING IT TOGETHER

“I played basketball from 14:00 to 27:00, and kept a log of it at /direct/phenix+u/beaumim/BASKETBALL.txt”

Regular Expression	What it Matches
[basketball]	Matches every letter in basketball, one letter at a time. Think of [] as a wild-card character, where you define the ‘wild cards’ inside.
[^basketball]	Matches every letter other than those in basketball, one letter at a time. This is the exactly inverted match, as the above match.
basketball	Matches every occurrence of basketball in all lowercase letters
[bB][aA][sS][kK][eE][tT][bB][aA][iL]{1,2}	Matches any case permutation of the string "basketball"
\d{1,2}:\d{1,2}	Matches any time-stamp which contains two 1 or 2 digit numbers separated with a colon.
/.*/	Matches any full unix-like directory.

Plug it in yourself to try it out at: <http://regexpal.com/>

SED – **STREAM EDITOR**

■ What is sed?

- A UNIX program (command) for modifying strings
- Primary use: string substitutions
 - Great place for regular expressions!

■ Syntax

- Note: sed syntax is the same as find/replace syntax in vi
- Applied to a file:

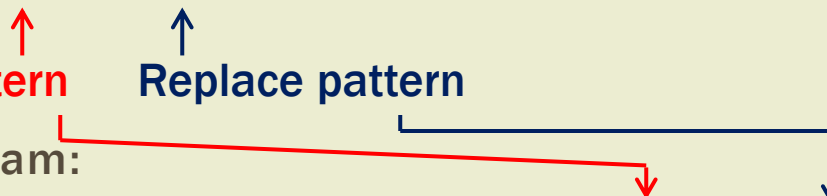
■ `$> sed 's/.../.../' <oldfile >newfile`

Find pattern **Replace pattern**



- Applied to a stream:

■ `$> echo "test string" | sed 's/.../.../'`



■ Detailed guide to sed:

- <http://www.grymoire.com/unix/sed.html>

SED: BASIC EXAMPLES

```
$> echo "test input" | sed 's/input/output/'  
test output
```

```
$> echo "test input" | sed 's/in/out/'  
test output
```

```
$> echo "test input" | sed 's/[ni]\{1,\}/out/'  
test output
```

Any character matching 'n' or 'i'
1 or more instances of the previous pattern

```
$> echo "test input" | sed 's/test input/"&"/'  
"test input"
```

& = all text matching pattern

```
$> echo "test input" | sed 's/\(test\) \(input\) "/"\2" "\1"/'  
"input" "test"
```

```
$> echo "test input" | sed 's/\([^ ]*\) \([a-z]*\) "/"\2" "\1"/'  
"input" "test"
```

Any not ' ' (space) character, zero or more
Any lower case letter character, zero or more

SED: ADVANCED EXAMPLE

```
readtree.C:
```

```
tree->SetBranchAddress("var1",&fill_var1);  
tree->SetBranchAddress("var2",&fill_var2);  
tree->SetBranchAddress("var3",&fill_var3);  
tree->SetBranchAddress("var4",&fill_var4);  
tree->SetBranchAddress("var5",&fill_var5);
```

```
$> sed 's/tree->SetBranchAddress("\(^[^"]*\)",&\([^\)]*\)/  
tree->Branch("\1",\&\2,"\1\F"/' <readtree.C >createtree.C
```

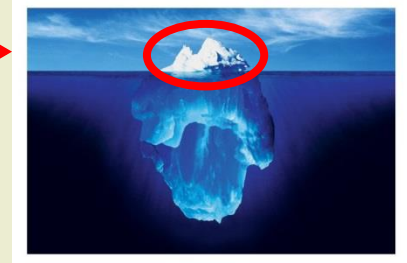
```
createtree.C:
```

```
tree->Branch("var1",&fill_var1,"var1/F");  
tree->Branch("var2",&fill_var2,"var2/F");  
tree->Branch("var3",&fill_var3,"var3/F");  
tree->Branch("var4",&fill_var4,"var4/F");  
tree->Branch("var5",&fill_var5,"var5/F");
```

AWK

■ What is awk?

- A versatile UNIX program (command) and scripting environment focused around processing string inputs
 - C style interpreter
- Great for processing strings in “column” form



■ Syntax

- ... | awk '{command1;command2;...;commandN}'

■ Possible commands:

- if (*conditional*) *statement* [else *statement*]
- while (*conditional*) *statement*
- for (*expression* ; *conditional* ; *expression*) *statement*
- for (*variable* in *array*) *statement*
- *variable*=*expression*
- print *expression*
- printf

■ Built in variables:

- \$0 – input line string
- \$1 – first ‘column’ of input string
 - \$n – n’t’h ‘column’ of input string
- FS – field separator character
 - awk -F: ‘{...}’ uses
- NF – number of input ‘columns’
- NR – current number of lines of input

AWK: EXAMPLES

```
$> ls -l | awk '{print "File \"$9\" is owned by \"$3\"a is \"$5}'}'  
File Desktop is owned by danielj and is 2.0K  
File README.txt is owned by danielj and is 120  
File core.2316 is owned by danielj and is 61M
```

```
$> ls -l | awk '{if($5>1000) print $9}'}'  
Desktop  
core.2316
```

```
$> hostname | awk '{if ($0 ~ /phenix.bnl/) print "1"; else if ($0  
~ /rcf.bnl/) print "2"; else print "3"}'
```

- Returns 1 on a phenix machine, 2 on an rcf machine, and 3 otherwise
- Useful for scripts: eg. Tell a script to give an error message if the above output = 1||3 if you only intend to run it on an rcf machine
- A Detailed awk guide: <http://www.grymoire.com/Unix/Awk.html>

END

- Tune in next week for part 2: scripting
 - See further examples of text manipulation in action!